

# TABELLE HASH

PIETRO DI LENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA  
UNIVERSITÀ DI BOLOGNA

ALGORITMI E STRUTTURE DI DATI  
ANNO ACCADEMICO 2021/2022



# INTRODUZIONE

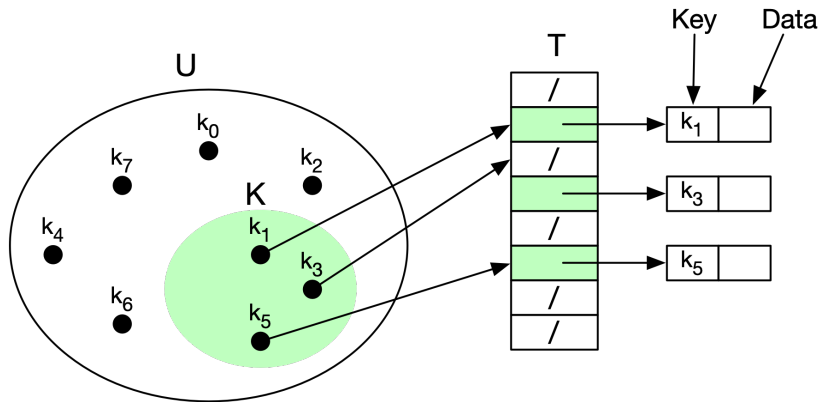
- Molte applicazioni richiedono una struttura dati di tipo Dizionario che supporti in maniera estremamente efficiente **unicamente** le operazioni basilari SEARCH, INSERT, DELETE
  - Esempio: i compilatori utilizzano un Dizionario per memorizzare ed etichettare gli identificatori (chiavi) nel programma
- La **Tabella Hash** implementa efficientemente la struttura dati Dizionario
  - **Idea**: generalizzare l'indicizzazione in un array ordinario
- Per quanto le operazioni su una Tabella Hash possano avere un costo pessimo lineare, **in media** le prestazioni computazionali sono efficienti
  - Sotto ragionevoli assunzioni probabilistiche le operazioni SEARCH, INSERT, DELETE hanno un costo medio  $O(1)$

# NOZIONI PRELIMINARI

- Indichiamo con
  - $U$  = Universo di **tutte le chiavi possibili**
  - $K$  = Insieme di **tutte le chiavi effettivamente utilizzate**
- Scelte implementative: dipendono dal dominio di applicazione
- Esempi:
  - $U = \{0, 1, \dots, m-1\}$  con  $m$  *piccolo*,  $|K| \sim |U|$ 
    - Usiamo **tabelle ad indirizzamento diretto**
  - $U$  è un insieme generico molto grande,  $|K| \ll |U|$ 
    - Usiamo **Tabelle Hash**

# TABELLE AD INDIRIZZAMENTO DIRETTO

- Implementazione basata su array  $T$  di dimensione  $|U|$
- La chiave  $k$  è memorizzata nella posizione  $k$  dell'array
- Ricordiamo che tutte le chiavi sono distinte



# TABELLE AD INDIRIZZAMENTO DIRETTO

```
1: function SEARCH(HASHTAB  $T$ , KEY  $k$ )  $\rightarrow$  DATA  
2:   return  $T[k].data$   
3:  
4: function INSERT(HASHTAB  $T$ , KEY  $k$ , DATA  $d$ )  
5:    $T[k] = \mathbf{new}$  NODE( $k, d$ )  
6:  
7: function DELETE(HASHTAB  $T$ , KEY  $k$ )  
8:   ERASE( $T[k]$ )  
9:    $T[k] = \mathbf{NIL}$ 
```

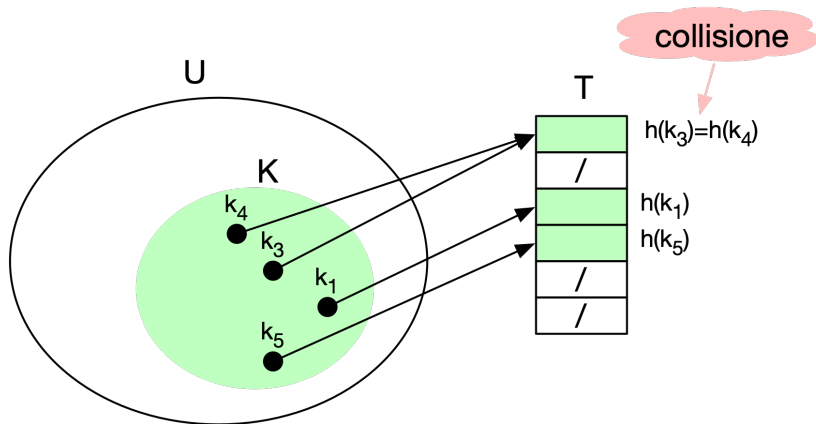
- Costo computazionale in termini di tempo:  $O(1)$
  - Costo computazionale in termini di memoria:  $\Theta(|T|) = \Theta(|U|)$ 
    - Se  $|K| \sim |U|$  soluzione accettabile
    - Se  $|K| \ll |U|$  soluzione non accettabile
      - Esempio:  $U$  = identificatori lunghi massimo 20 caratteri
- $|U| > 26 * (26 + 10)^{19} \approx 10^{31} \Rightarrow |T| \approx 10^{31} * 4\text{bytes} > 10^{19}\text{Terabytes}$

# TABELLE HASH

- Un array di dimensione  $\Theta(|U|)$  richiede troppa memoria se  $U$  è grande
- Generalmente l'insieme di chiavi  $K$  è molto più piccolo rispetto ad  $U$
- **Soluzione:** Tabelle Hash
  - Usiamo un array  $T[0, \dots, m-1]$  di dimensione  $m = \Theta(|K|)$
  - Usiamo una **funzione hash**  $h : U \rightarrow [0, \dots, m-1]$
- Indirizzamento hash
  - Diciamo che  $h(k)$  è il **valore hash** della chiave  $k$
  - La funzione  $h$  trasforma una chiave  $k$  in un indice dell'array  $T$
  - La chiave  $k$  viene *mappata* nello slot  $T[h(k)]$
  - Se due chiavi hanno lo stesso valore hash abbiamo una **collisione**
- **Problema:** evitare e gestire le collisioni hash
  - Idealmente vorremmo funzioni hash che evitino sempre collisioni
  - Non possiamo evitarle, dobbiamo almeno minimizzarle

# TABELLE HASH

- Implementazione basata su array  $T$  di dimensione  $\Theta(|K|)$
- La chiave  $k$  e i dati sono memorizzati nella posizione  $h(k)$  dell'array
- Evitare le collisioni è impossibile anche con buone funzioni hash



- Per implementare una Tabella Hash efficiente abbiamo bisogno di
  - 1 Una funzione hash
    - Deve poter essere calcolata velocemente
    - Deve garantire una buona distribuzione delle chiavi su  $T$
    - Una buona distribuzione minimizza il rischio di collisioni
  - 2 Un metodo per gestire le collisioni
    - Le collisioni sono inevitabili
    - Quando non riusciamo ad evitarle dobbiamo gestirle
  - 3 Un vettore  $T[0, \dots, m-1]$  di dimensione  $m = \Theta(|K|)$ 
    - Normalmente possiamo solo stimare  $m$
    - Non sappiamo a priori quante chiavi andremo a memorizzare
    - Potrebbe essere necessario ridimensionare  $T$
    - La scelta migliore per la dimensione  $m$  dipende dalla funzione hash e dal metodo utilizzato per gestire le collisioni



# FUNZIONI HASH

- Una buona funzione hash soddisfa (approssimativamente) la proprietà di **uniformità semplice** (*hashing uniforme semplice*)
  - Una funzione hash  $h$  deve distribuire uniformemente le chiavi negli indici  $[0, \dots, m - 1]$  della tabella  $T$
  - Ogni indice  $i = h(k)$  deve essere generato con probabilità  $1/m$
  - Se alcuni indici in  $[0, \dots, m - 1]$  sono *scelti* con maggiore probabilità da  $h$  allora avremo un numero maggiore di collisioni
- Per soddisfare la proprietà di uniformità semplice bisogna conoscere la distribuzione di probabilità con cui le chiavi sono *estratte* da  $U$ 
  - Conoscere tale distribuzione di probabilità è spesso irrealistico
  - Solo in casi specifici tale distribuzione è nota
  - Esempio: assumendo che le chiavi  $k$  siano estratte **a caso** da  $U = [0, 1)$  (tutte le chiavi in  $U$  sono equiprobabili). Allora

$$h(k) = \lfloor mk \rfloor$$

soddisfa la proprietà di uniformità semplice

# FUNZIONI HASH: ASSUNZIONI

## 1 Tutte le chiavi sono equiprobabili

- Tutte le chiavi hanno la stessa probabilità di essere estratte da  $U$
- Non è sempre vero (es. identificatori in un programma)
- Semplificazione necessaria per proporre un meccanismo generale

## 2 La funzione hash può essere calcolata in tempo $O(1)$

- Una codifica hash non  $O(1)$  domina il costo delle operazioni
- Es. costa più calcolare il valore hash che effettuare una ricerca
- In realtà, ci accontentiamo di hashing sufficientemente veloci

## 3 Tutte le chiavi sono valori interi non-negativi

- E' sempre possibile trasformare una qualsiasi chiave  $k$  in un intero
- Es. numero decimale ottenuto dalla rappresentazione binaria di  $k$

## ESEMPIO: DA STRINGA AD INTERO POSITIVO

- Vogliamo trasformare una chiave di tipo *stringa* in intero
- Idea: trasformiamo i caratteri in un codice binario
- Assumiamo di associare i seguenti codici alle lettere dell'alfabeto

$$a = 1, b = 2, c = 3, d = 4, e = 5 \dots, r = 18, \dots z = 26$$

in binario sono sufficienti 6 bit per carattere

- Codifica ottenuta concatenando i codici binari

$$\text{bin}(\text{"beer"}) = 000010\ 000101\ 000101\ 010010$$

che rappresenta il numero 545.106 in base 10

# FUNZIONE HASH: METODO DELLA DIVISIONE

- Metodo della divisione:  $h(k) = k \bmod m$
- Esempi:
  - Se  $m = 12, k = 100 \Rightarrow h(k) = 4$
  - Se  $m = 10, k = 101 \Rightarrow h(k) = 1$
- Vantaggi:
  - Molto efficiente (richiede solo una divisione intera)
- Svantaggi:
  - Suscettibile a specifici valori di  $m$  (potrebbe non usare tutto  $k$ )
  - Esempio 1: se  $m = 10$  allora  $h(k)$  = ultima cifra di  $k$
  - Esempio 2: se  $m = 2^p$  allora  $h(k)$  dipende unicamente dai  $p$  bit meno significativi di  $k$  e non da tutti i bit di  $k$
  - Soluzione: scegliere  $m$  come numero primo distante da potenze di 2 (e di 10)

# FUNZIONE HASH: METODO DELLA MOLTIPLICAZIONE

- **Metodo della moltiplicazione:**  $h(k) = \lfloor m(kC - \lfloor kC \rfloor) \rfloor$ 
  - Sia  $C$  una costante  $0 < C < 1$
  - Moltiplichiamo  $k$  per  $C$  e prendiamo la parte frazionaria
  - Moltiplichiamo quest'ultima per  $m$  e prendiamo la parte intera
- **Esempi:**
  - Se  $m = 12, k = 101, C = 0.8 \Rightarrow h(k) = 9$
  - Se  $m = 1000, k = 124, C = (\sqrt{5} - 1)/2 \approx 0.618 \Rightarrow h(k) = 18$
- **Svantaggi:**
  - La costante  $C$  influenza la proprietà di uniformità di  $h$
  - $C = (\sqrt{5} - 1)/2$  suggerito da Knuth (*The Art of Computer Programming, Vol 3*)
- **Vantaggi:**
  - Il valore di  $m$  non è critico

# FUNZIONE HASH: METODO DELLA CODIFICA ALGEBRICA

## ■ Metodo della codifica algebrica:

$$h(k) = (k_n x^n + k_{n-1} x^{n-1} + \dots + k_1 x + k_0) \bmod m$$

dove

- $k = k_n k_{n-1} \dots k_1 k_0$  e  $k_i$  è l' $i$ -esimo bit della rappresentazione binaria di  $k$ , oppure l' $i$ -esima cifra della rappresentazione decimale di  $k$ , o anche il codice ascii dell' $i$ -esimo carattere
- $x$  è un valore costante
- Esempio: usando la rappresentazione decimale
  - Se  $m=12, k=234, x=3 \Rightarrow h(k) = (2 \times 3^2 + 3 \times 3 + 4) \bmod 12 = 7$
- Vantaggi:
  - Dipende da tutti i bit/caratteri della chiave
- Svantaggi:
  - Costoso da calcolare
  - Richiede  $n$  addizioni e  $n * (n + 1)/2$  prodotti ( $n = \Theta(\log k)$ )

# REGOLA DI HORNER

- Valutazione di un polinomio in un punto
- Un polinomio di grado  $n$

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

può essere riscritto nel seguente modo

$$p(x) = a_0 + x(a_1 + x(a_2 + x(\cdots x(a_{n-1} + a_n x))))$$

che richiede  $n$  addizioni ed  $n$  moltiplicazioni

- La regola di Horner permette di abbassare il costo (da quadratico a lineare sul numero di cifre della chiave  $k$ ) del calcolo della funzione hash basata sul metodo della codifica algebrica
  - Dato che il numero di cifre di una chiave è tipicamente un numero relativamente piccolo, possiamo assumere un costo costante per il metodo della codifica

# JAVA.LANG.STRING.HASHCODE()

```
/**
 * Returns a hash code for this string. The hash code for a String
 * object is computed as
 *
 *  $s[0]*31^{(n-1)} + s[1]*31^{(n-2)} + \dots + s[n-1]$ 
 *
 * using int arithmetic, where  $s[i]$  is the  $i$ th character of the
 * string,  $n$  is the length of the string, and  $^$  indicates
 * exponentiation. (The hash value of the empty string is zero.)
 */
public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
        char val[] = value;

        for (int i = 0; i < value.length; i++)
            h = 31 * h + val[i];
        hash = h;
    }
    return h;
}
```

- Funzione hash di libreria Java della classe String
- Basata sul metodo della codifica algebrica
  - Utilizza i codici ascii dei caratteri
  - Calcolata con il metodo di Horner
  - La costante  $x$  è il numero primo 31



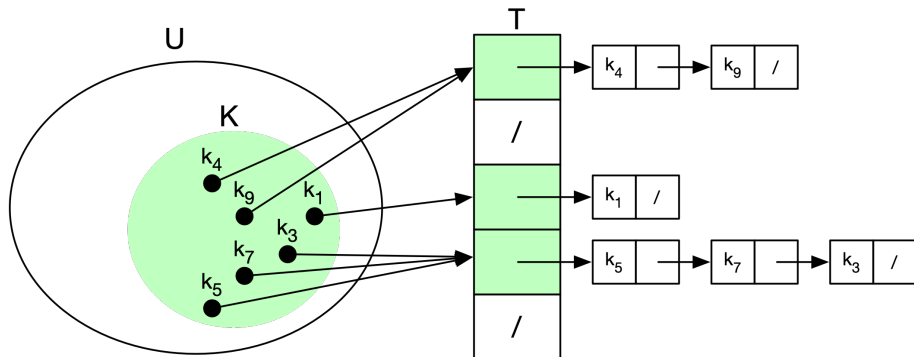
# PROBLEMA DELLE COLLISIONI

- Hashing uniforme semplice riduce ma non elimina le collisioni
- Anche assumendo hashing uniforme semplice, la probabilità che ci sia collisione tra due chiavi è (sorprendentemente) molto alta
  - Problema del compleanno: date  $n$  persone scelte a caso qual è la probabilità che due tra esse compiano gli anni nello stesso giorno?
  - **Paradosso del compleanno**: in un gruppo di 23 persone tale probabilità è maggiore del 50%
- Come gestire le eventuali collisioni?
  - Dobbiamo trovare collocazioni alternative per le chiavi
  - Se una chiave non si trova nella posizione attesa, bisogna andare a cercare nelle posizioni alternative
  - Le operazioni diventano costose nel caso pessimo
- Vediamo due possibili tecniche
  - **Concatenamento** (o anche **scansione esterna**)
  - **Indirizzamento aperto** (o anche **scansione interna**)

# RISOLUZIONE COLLISIONI: CONCATENAMENTO

## ■ Concatenamento (chaining)

- Le chiavi  $k$  con lo stesso valore hash  $h(k) = i$  sono memorizzate in una lista concatenata (**lista di trabocco**)
- Lo slot  $T[i]$  contiene il puntatore alla testa della lista contenente tutte le chiavi  $k$  con hash  $h(k) = i$



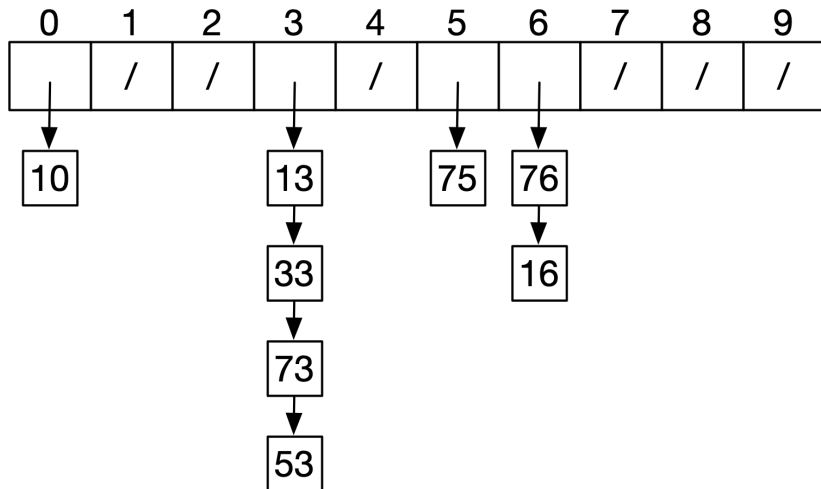
# PSEUDOCODICE: CONCATENAMENTO

```
1: function SEARCH(HASHTAB  $T$ , KEY  $k$ )  $\rightarrow$  DATA
2:    $tmp = \text{LINKED-LIST-SEARCH}(T[h(k)], k)$ 
3:   if  $tmp \neq \text{NIL}$  then
4:     return  $tmp.data$ 
5:   else
6:     return NIL
7:
8: function INSERT(HASHTAB  $T$ , KEY  $k$ , DATA  $d$ )
9:   LINKED-LIST-INSERT( $T[h(k)], k, d$ )
10:
11: function DELETE(HASHTAB  $T$ , KEY  $k$ )
12:    $T[h(k)] = \text{LINKED-LIST-DELETE}(T[h(k)], k)$ 
```

- SEARCH esegue una ricerca lineare su lista concatenata
- INSERT esegue un inserimento in testa in una lista concatenata
- DELETE esegue una rimozione su una lista concatenata

# ESEMPIO: CONCATENAMENTO

- Funzione hash  $h(k) = k \bmod 10$
- Inserimenti nel seguente ordine: 53, 75, 16, 73, 10, 33, 13, 76



# ANALISI DEL METODO DI CONCATENAMENTO

- Dimensione della tabella
  - Non impone vincoli sulla dimensione del vettore  $T[0, \dots, m-1]$ 
    - Vincoli eventualmente imposti dalla funzione hash
  - Se  $m$  troppo grande, rischio di sprecare spazio
  - Se  $m$  troppo piccolo, liste di collisione lunghe  $\Rightarrow$  nel caso pessimo operazione di ricerca di una chiave ha un costo lineare sulla dimensione della lista
- Quanto costano le operazioni SEARCH, INSERT, DELETE?
  - Sia  $L$  la lunghezza della lista di collisione più lunga
  - SEARCH: costo nel caso pessimo  $O(L)$ , caso ottimo  $O(1)$
  - INSERT: costo nel caso pessimo e ottimo  $O(1)$
  - DELETE: costo nel caso pessimo  $O(L)$ , caso ottimo  $O(1)$
  - N.B.  $L = O(n)$ , dove  $n$  = numero di elementi nella tabella
  - N.B. Il costo pessimo non dipende da  $m$  ma dal numero di elementi  $n$
  - Riusciamo a analizzare il caso medio?

# CONCATENAMENTO: ANALISI DEL CASO MEDIO

- Il costo nel **caso medio** dipende dal **numero medio di accessi** per cercare (con successo o insuccesso) una chiave
  - Il costo della ricerca di una chiave incide su SEARCH e DELETE
  - Il numero medio di accessi dipende da come vengono distribuite le chiavi dalla funzione hash
- Chiamiamo **fattore di carico**  $\alpha = n/m$  il rapporto tra il numero di elementi e la dimensione di una Tabella Hash
  - $n$  = numero di elementi nella Tabella Hash
  - $m$  = numero di slot nella Tabella Hash
- Sotto l'assunzione di **hashing uniforme semplice** ogni slot della tabella ha **mediamente**  $\alpha$  chiavi
  - Ricordiamo che **hashing uniforme semplice**  $\Rightarrow$  la funzione hash distribuisce le chiavi uniformemente in  $T[0, \dots, m-1]$

## Teorema

Sotto l'assunzione di hashing uniforme semplice, una ricerca senza successo in una tabella hash con concatenamento ha costo medio  $\Theta(1 + \alpha)$

### ■ Dimostrazione

- Sotto l'assunzione di hashing uniforme semplice, data una chiave  $k$  non presente nella tabella, gli  $m$  slot di  $T$  sono tutti ugualmente probabili per la codifica hash  $h(k)$
- Se  $k$  non compare nella tabella (ricerca con insuccesso), la ricerca visita tutte le chiavi nella lista  $T[h(k)]$ , che ha in media  $\alpha$  chiavi
- **Costo medio: tempo di hashing  $h(k)$  (costo medio 1) + tempo di visita della lista  $T[h(k)]$  (costo medio  $\alpha$ )  $\Rightarrow \Theta(1 + \alpha)$**

## Teorema

Sotto l'assunzione di hashing uniforme semplice, una ricerca con successo in una tabella hash con concatenamento ha costo medio  $\Theta(1 + \alpha)$

### ■ Dimostrazione

- Sotto l'assunzione di hashing uniforme semplice, data una chiave  $k$  non presente nella tabella, gli  $m$  slot di  $T$  sono tutti ugualmente probabili per la codifica hash  $h(k)$
- Se  $k$  compare nella tabella (ricerca con successo), la ricerca **visita in media** all'incirca metà delle chiavi nella lista  $T[h(k)]$  (l'analisi completa è più complessa), che ha in media  $\alpha$  chiavi
- **Costo medio: tempo di hashing**  $h(k)$  (costo medio 1) + **tempo di visita della lista**  $T[h(k)]$  (costo medio  $\alpha/2$ )  $\Rightarrow$   
$$\Theta(1 + \alpha/2) = \Theta(1 + \alpha)$$

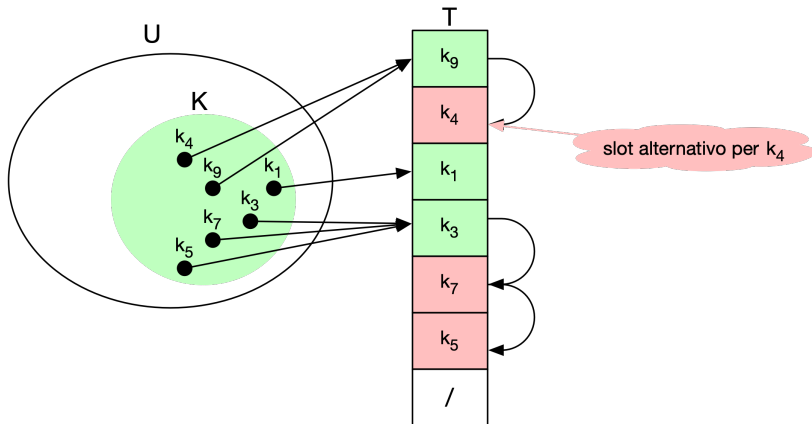


# RIASSUMENDO: ANALISI DEL CASO MEDIO

- Abbiamo dimostrato che su una Tabella Hash in cui le collisioni siano risolte con concatenamento, sotto l'assunzione di hashing uniforme semplice, la ricerca ha un **costo medio**  $\Theta(1 + \alpha)$ 
  - $n$  = numero di elementi nella tabella
  - $m$  = numero di slot nella tabella
  - fattore di carico  $\alpha = n/m$
- Il **fattore di carico** influenza quindi il costo delle operazioni
  - Se  $n = O(m)$  allora  $\alpha = O(1) \Rightarrow$  **costo medio della ricerca**  $O(1)$
  - Quindi SEARCH, INSERT, DELETE hanno costo medio  $O(1)$

# RISOLUZIONE COLLISIONI: INDIRIZZAMENTO APERTO

- Indirizzamento aperto (open addressing)
  - Tutte le chiavi sono memorizzate nella stessa tabella
  - Ogni slot contiene una chiave oppure NIL
  - Se uno slot è occupato, se ne cerca uno alternativo nella tabella



# INDIRIZZAMENTO APERTO: ISPEZIONI

- **Idea:** data una chiave  $k$ , se uno slot  $T[h(k)]$  è già occupato allora **ispezioniamo** la tabella alla ricerca di uno slot libero
- Per determinare quale slot ispezionare estendiamo la funzione hash in modo che abbia come parametro anche il **passo di ispezione**

$$h : U \times [0, \dots, m-1] \rightarrow [0, \dots, m-1]$$

- La **sequenza di ispezione**

$$h(k, 0), h(k, 1), \dots, h(k, m-1)$$

deve fornire una **permutazione** degli indici della tabella

- Vogliamo visitare ogni slot solo una volta
- Potrebbe essere necessario visitare tutti gli  $m$  slot

# PSEUDOCODICE: INDIRIZZAMENTO APERTO

```
1: function SEARCH(HASHTAB  $T$ , KEY  $k$ )  $\rightarrow$  DATA
2:    $i = 0$ 
3:   repeat
4:      $j = h(k, i)$  ▷ hash value at step  $i$ 
5:     if  $A[j].key == k$  then
6:       return  $A[j].data$ 
7:      $i = i + 1$ 
8:   until  $A[j] == \text{NIL}$  or  $i == A.size$ 
9:   return NIL
```

```
1: function INSERT(HASHTAB  $T$ , KEY  $k$ , DATA  $d$ )
2:    $i = 0$ 
3:   repeat
4:      $j = h(k, i)$  ▷ hash value at step  $i$ 
5:     if  $A[j] == \text{NIL}$  then
6:        $A[j] = (k, d)$ 
7:       return
8:      $i = i + 1$ 
9:   until  $i == A.size$ 
10:  error "overflow"
```

Attenzione: non funziona correttamente

# PSEUDOCODICE DELETE: INDIRIZZAMENTO APERTO

- Non possiamo sostituire la chiave che vogliamo cancellare con NIL
  - SEARCH si ferma se trova NIL mentre la chiave cercata potrebbe essere presente e verrebbe trovata nelle ispezioni successive
- Soluzione: utilizziamo il valore DELETED invece di NIL per marcare uno slot vuoto dopo la cancellazione
  - SEARCH/DELETE: DELETED trattati come slot pieni
  - INSERT: DELETED trattati come slot vuoti

```
1: function DELETE(HASHTAB  $T$ , KEY  $k$ )
2:    $i = 0$ 
3:   repeat
4:      $j = h(k, i)$  ▷ hash value at step  $i$ 
5:     if  $A[j].key == k$  then
6:       ERASE( $A[j]$ )
7:        $A[j] = \text{DELETED}$ 
8:     return
9:      $i = i + 1$ 
10:  until  $A[j] == \text{NIL}$  or  $i == A.size$ 
```

# PSEUDOCODICE: INDIRIZZAMENTO APERTO

```
1: function SEARCH(HASHTAB  $T$ , KEY  $k$ )  $\rightarrow$  DATA
2:    $i = 0$ 
3:   repeat
4:      $j = h(k, i)$   $\triangleright$  hash value at step  $i$ 
5:     if  $A[j].key == k$  then
6:       return  $A[j].data$ 
7:      $i = i + 1$ 
8:   until  $A[j] == \text{NIL}$  or  $i == A.size$ 
9:   return NIL
```

```
1: function INSERT(HASHTAB  $T$ , KEY  $k$ , DATA  $d$ )
2:    $i = 0$ 
3:   repeat
4:      $j = h(k, i)$   $\triangleright$  hash value at step  $i$ 
5:     if  $A[j] == \text{NIL}$  or  $A[j] == \text{DELETED}$  then
6:        $A[j] = (k, d)$ 
7:     return
8:      $i = i + 1$ 
9:   until  $i == A.size$ 
10:  error "overflow"
```

# ANALISI DEL METODO DI INDIRIZZAMENTO APERTO

- Nel caso pessimo SEARCH, INSERT, DELETE costano  $O(m)$ 
  - $m$  = dimensione della tabella
  - Nel caso pessimo ispezioniamo l'intera tabella
- Quanto costano le operazioni nel caso medio?
  - Costo medio influenzato dalla strategia di ispezione
  - Anche sotto l'assunzione di hashing uniforme semplice
- Vediamo tre strategie di ispezione
  - ispezione lineare
  - ispezione quadratica
  - doppio hashing

# STRATEGIE DI ISPEZIONE: ISPEZIONE LINEARE

- Funzione di ispezione ( $m$  = dimensione della tabella)

$$h(k, i) = (h'(k) + i) \bmod m$$

dove  $h'(k)$  è una funzione hash ausiliaria

- Quando si ha una collisione, si ispeziona l'indice successivo

- Il primo indice  $h'(k)$  determina l'intera sequenza

$$h'(k), h'(k) + 1, \dots, m - 1, 0, 1, \dots, h'(k) - 1$$

- Sono possibili solo  $m$  sequenze distinte di ispezione
  - Ogni slot è ispezionato una sola volta

- Problema: **clustering primario**

- Lunghe sotto-sequenze occupate, che diventano sempre più lunghe
  - Assumendo hashing uniforme semplice, uno slot vuoto preceduto da  $i$  slot pieni viene riempito con probabilità  $(i + 1)/m$
  - I tempi medi di inserimento e cancellazione crescono



# ESEMPIO: ISPEZIONE LINEARE

- Funzione hash  $h(k, i) = (h'(k) + i) \bmod 10$  dove  $h'(k) = k \bmod 10$
- Inserimenti nel seguente ordine: 53, 75, 16, 73, 10, 33, 13, 76

insert 53	0	1	2	3	4	5	6	7	8	9	$h(53,0) = 3$
	/	/	/	53	/	/	/	/	/	/	
insert 75	0	1	2	3	4	5	6	7	8	9	$h(75,0) = 5$
	/	/	/	53	/	75	/	/	/	/	
insert 16	0	1	2	3	4	5	6	7	8	9	$h(16,0) = 6$
	/	/	/	53	/	75	16	/	/	/	
insert 73	0	1	2	3	4	5	6	7	8	9	$h(73,1) = 4$ $h(73) = 3$
	/	/	/	53	73	75	16	/	/	/	
insert 10	0	1	2	3	4	5	6	7	8	9	$h(10,0) = 0$
	10	/	/	53	73	75	16	/	/	/	
insert 33	0	1	2	3	4	5	6	7	8	9	$h(33,4) = 7$ $h(33) = 3$
	10	/	/	53	73	75	16	33	/	/	
insert 13	0	1	2	3	4	5	6	7	8	9	$h(13,5) = 8$ $h(13) = 3$
	10	/	/	53	73	75	16	33	13	/	
insert 76	0	1	2	3	4	5	6	7	8	9	$h(76,3) = 9$ $h(76) = 6$
	10	/	/	53	73	75	16	33	13	76	

# ESEMPIO: ISPEZIONE LINEARE

$h(k, i) = (h'(k) + i) \bmod 31$  dove  $h'(k) = \text{ascii}(k) \bmod 31$  ( $\text{ascii}(A) = 65$ )

	C E I L M N O P R S T V																																							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30									
P																				P																				
R																			P	R																				
E							E												P	R																				
C						C	E												P	R																				
I						C	E				I								P	R																				
P						C	E				I								P	P	R																			
I						C	E				I	I							P	P	R																			
T						C	E				I	I							P	P	R										T									
E						C	E	E			I	I							P	P	R										T									
V						C	E	E			I	I							P	P	R										V									
O						C	E	E			I	I							O	P	P	R									T	V								
L						C	E	E			I	I			L				O	P	P	R									T	V								
I						C	E	E			I	I	I		L				O	P	P	R									T	V								
S						C	E	E			I	I	I		L				O	P	P	R									S	T	V							
S						C	E	E			I	I	I		L				O	P	P	R									S	T	S	V						
I						C	E	E			I	I	I		L				O	P	P	R									S	T	S	V						
M						C	E	E			I	I	I		L				O	P	P	R									S	T	S	V						
E						C	E	E	E		I	I	I		L				O	P	P	R									S	T	S	V						
V						C	E	E	E		I	I	I		L				O	P	P	R									S	T	S	V	V					
O						C	E	E	E		I	I	I		L				O	P	P	R									S	T	S	V	V	O				
L						C	E	E	E		I	I	I		L				O	P	P	R									S	T	S	V	V	O	L			
M						C	E	E	E		I	I	I		L				O	P	P	R									S	T	S	V	V	O	L	M		
E						C	E	E	E	E		I	I	I		L			O	P	P	R									S	T	S	V	V	O	L	M		
N						C	E	E	E	E		I	I	I		L			O	P	P	R									S	T	S	V	V	O	L	M	N	
T						C	E	E	E	E		I	I	I		L			O	P	P	R									S	T	S	V	V	O	L	M	N	T
E	E					C	E	E	E	E		I	I	I		L			O	P	P	R									S	T	S	V	V	O	L	M	N	T
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30									

# STRATEGIE DI ISPEZIONE: ISPEZIONE QUADRATICA

- Funzione di ispezione ( $m =$  dimensione della tabella)

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \quad (\text{con costanti } c_1 \neq c_2)$$

dove  $h'(k)$  è una funzione hash ausiliaria

- Quando si ha una collisione, si usa un passo quadratico
  - Il primo indice  $h'(k)$  determina l'intera sequenza
  - Le ispezioni successive hanno un offset che dipende da una funzione quadratica nel numero di ispezione  $i$
  - Sono possibili solo  $m$  sequenze distinte di ispezione
  - $c_1, c_2$  devono garantire una permutazione di  $[0, \dots, m-1]$
- Problema: **clustering secondario**
  - Se due chiavi hanno la stessa ispezione iniziale, allora le loro sequenze di ispezione sono identiche

# ESEMPIO: ISPEZIONE QUADRATICA

- Funzione hash  $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod 10$  dove  
 $h'(k) = k \bmod 10$  e  $c_1 = 0, c_2 = 1$
- Inserimenti nel seguente ordine: 53, 75, 16, 73, 10, 33, 13, 76

	0	1	2	3	4	5	6	7	8	9	
insert 53	/	/	/	53	/	/	/	/	/	/	$h(53,0) = 3$
	0	1	2	3	4	5	6	7	8	9	
insert 75	/	/	/	53	/	75	/	/	/	/	$h(75,0) = 5$
	0	1	2	3	4	5	6	7	8	9	
insert 16	/	/	/	53	/	75	16	/	/	/	$h(16,0) = 6$
	0	1	2	3	4	5	6	7	8	9	
insert 73	/	/	/	53	73	75	16	/	/	/	$h(73,1) = 4$ $h'(73) = 3$
	0	1	2	3	4	5	6	7	8	9	
insert 10	10	/	/	53	73	75	16	/	/	/	$h(10,0) = 0$
	0	1	2	3	4	5	6	7	8	9	
insert 33	10	/	/	53	73	75	16	33	/	/	$h(33,2) = 7$ $h'(33) = 3$
	0	1	2	3	4	5	6	7	8	9	
insert 13	10	/	13	53	73	75	16	33	/	/	$h(13,3) = 2$ $h'(13) = 3$
	0	1	2	3	4	5	6	7	8	9	
insert 76	10	76	13	53	73	75	16	33	/	/	$h(76,5) = 1$ $h'(76) = 6$

# STRATEGIE DI ISPEZIONE: DOPPIO HASHING

- Funzione di ispezione ( $m$  = dimensione della tabella)

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

dove  $h_1(k)$  e  $h_2(k)$  sono la funzione hash primaria e secondaria

- Quando si ha una collisione, si usa la funzione secondaria e l'indice di ispezione per determinare il successivo slot da ispezionare

- Evita il clustering primario e secondario

- Se  $h_1 \neq h_2$  è meno probabile che per una coppia di chiavi  $a \neq b$

$$h_1(a) = h_1(b) \text{ e } h_2(a) = h_2(b)$$

- Sono possibili più di  $m$  sequenze distinte di ispezione

- Vincoli sulla funzione hash secondaria  $h_2$

- Non deve mai dare il valore hash 0

- Deve permettere di iterare su tutta la tabella

# ESEMPIO: DOPPIO HASHING

- Funzione hash  $h(k, i) = (h_1(k) + ih_2(k)) \bmod 10$  dove  
 $h_1(k) = k \bmod 10$  e  $h_2(k) = (k \bmod 9) + 1$
- Inserimenti nel seguente ordine: 53, 75, 16, 73, 10, 33, 13, 76

	0	1	2	3	4	5	6	7	8	9	
insert 53	/	/	/	53	/	/	/	/	/	/	$h(53,0) = 3$
	0	1	2	3	4	5	6	7	8	9	
insert 75	/	/	/	53	/	75	/	/	/	/	$h(75,0) = 5$
	0	1	2	3	4	5	6	7	8	9	
insert 16	/	/	/	53	/	75	16	/	/	/	$h(16,0) = 6$
	0	1	2	3	4	5	6	7	8	9	
insert 73	/	/	/	53	/	75	16	73	/	/	$h(73,2) = 7$ $h_2(73) = 2$
	0	1	2	3	4	5	6	7	8	9	
insert 10	10	/	/	53	/	75	16	73	/	/	$h(10,0) = 0$
	0	1	2	3	4	5	6	7	8	9	
insert 33	10	/	/	53	33	75	16	73	/	/	$h(33,3) = 4$ $h_2(33) = 7$
	0	1	2	3	4	5	6	7	8	9	
insert 13	10	/	/	53	33	75	16	73	13	/	$h(13,1) = 8$ $h_2(13) = 5$
	0	1	2	3	4	5	6	7	8	9	
insert 76	10	76	/	53	33	75	16	73	13	/	$h(76,1) = 1$ $h_2(76) = 5$

# ESEMPIO: DOPPIO HASHING

$h(k, i) = (h_1(k) + ih_2(k)) \bmod 31$  dove

$h_1(k) = \text{ascii}(k) \bmod 31$  e  $h_2(k) = (h_1(k) \bmod 30) + 1$

	C E I L M N O P R S T V																															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	
P																			P													
R																			P	R												
E						C	E												P	R												
I					C	E					I								P	R												
P					C	P	E				I								P	R												
I					C	P	E				I								P	R												
T					C	P	E				I								P	R					T	I						
E					C	P	E				I					E			P	R				T	I							
V					C	P	E				I					E			P	R				T	I	V						
O					C	P	E				I					E		O	P	R				T	I	V						
L					C	P	E				I				L	E		O	P	R				T	I	V						
I					I	C	P	E			I				L	E		O	P	R				T	I	V						
S					I	C	P	E			I				L	E		O	P	R				S	T	I	V					
S					I	C	P	E			I				L	E		O	P	R				S	T	I	V					
I					I	C	P	E			I				L	E		I	O	P	R			S	T	I	V					
M	M				I	C	P	E			I				L	E		I	O	P	R			S	T	I	V					
E	M				I	C	P	E	E		I				L	E		I	O	P	R			S	T	I	V					
V	M				I	C	P	E	E		I				L	E		I	O	P	R			S	T	I	V	V				
O	M				I	C	P	E	E	O		I			L	E		I	O	P	R			S	T	I	V	V				
L	M				I	C	P	E	E	O		I			L	E		I	O	P	R			S	T	I	V	V				L
M	M	M			I	C	P	E	E	O		I			L	E		I	O	P	R			S	T	I	V	V				L
N	M	M	E		I	C	P	E	E	O		I			L	E		I	O	P	N	R		S	T	I	V	V				L
T	M	M	E		I	C	P	E	E	O		I			S	T	L	E	I	O	P	N	R		S	T	I	V	V			L
E	M	M	E		I	C	P	E	E	O	E	I	S	T	L	E		I	O	P	N	R		S	T	I	V	V				L
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	

# INDIRIZZAMENTO APERTO: ANALISI DEL CASO MEDIO

- Utilizziamo il fattore di carico  $\alpha = n/m$  anche per l'analisi del costo medio col metodo di indirizzamento aperto
  - In questo caso, poiché  $n \leq m$ , abbiamo che  $\alpha < 1$
- Assumiamo **hashing uniforme semplice**
- Assumiamo inoltre che le **permutazioni degli indici**  $[0, \dots, m-1]$  determinate dalle sequenze di ispezione

$$h(k, 0), h(k, 1), \dots, h(k, m-1)$$

**siano tutte equiprobabili**

- Ogni chiave  $k$  ha un'unica sequenza di ispezione associata
- Ogni sequenza di ispezioni è ugualmente probabile
- Questo dipende dalla strategia di ispezione: **l'ispezione lineare non soddisfa tale proprietà**, mentre è soddisfatta dall'ispezione quadratica e doppio hashing



# INDIRIZZAMENTO APERTO: ANALISI DEL CASO MEDIO

## Teorema (ricerca senza successo)

Sotto l'assunzione di hashing uniforme semplice e sequenze di ispezione equiprobabili, il numero medio di ispezioni di una ricerca senza successo in una tabella hash con indirizzamento aperto e fattore di carico  $\alpha < 1$  è al massimo  $1/(1 - \alpha)$

## Teorema (ricerca con successo)

Sotto l'assunzione di hashing uniforme semplice e sequenze di ispezione equiprobabili, il numero medio di ispezioni di una ricerca con successo in una tabella hash con indirizzamento aperto e fattore di carico  $\alpha < 1$  è al massimo  $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

- In entrambi i casi, se  $\alpha$  è costante il tempo di accesso è  $O(1)$
- Se la tabella è piena al 50%, la ricerca senza successo richiede in media al massimo due ispezioni, la ricerca con successo meno di due
- Se la tabella è piena al 90%, la ricerca senza successo richiede in media al massimo dieci ispezioni, la ricerca con successo meno di tre

# ANALISI DEL CASO MEDIO: ISPEZIONE LINEARE

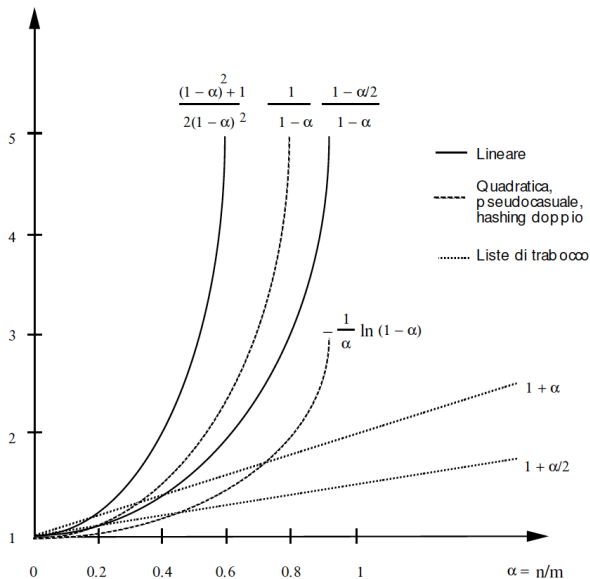
- L'**ispezione lineare** non assicura che le sequenze di ispezione siano tutte equiprobabili (effetto del clustering primario)
- Il costo medio non è caratterizzato dai due teoremi precedenti
  - Il costo medio nel caso di ricerca senza successo è al massimo

$$\frac{(1 - \alpha)^2 + 1}{2(1 - \alpha)^2}$$

- Il costo medio nel caso di ricerca con successo è al massimo

$$\frac{(1 - \alpha/2)}{2(1 - \alpha)}$$

# CONFRONTO COSTI MEDI DI ISPEZIONE



# COMMENTI GENERALI: RUOLO DEL FATTORE DI CARICO

- Le prestazioni delle tabelle hash sono legate al fattore di carico  $\alpha$
- Secondo il **paradosso del compleanno**, le collisioni sono molto probabili
  - Le collisioni sono praticamente inevitabili anche su sottoinsiemi relativamente piccoli di possibili chiavi
- Strategia: mantenere il fattore di carico basso
  - Un fattore di carico  $\alpha < 0.75$  è considerato ottimale
  - Ridimensioniamo la tabella quando il fattore di carico supera una certa soglia critica
  - N.B. Ridimensionare la completa ricostruzione della Tabella Hash poiché gli indici hash cambiano

# TABELLE HASH IN JAVA

## ■ `JAVA.UTIL.HASHMAP`

- classe equivalente a `HASHTABLE` (che è sincronizzata)
- Gestione delle collisioni con concatenamento
- Java 7: liste di trabocco con liste concatenate
- Java 8: liste di trabocco con RB alberi (alberi bilanciati di ricerca)
  - Costo pessimo logaritmico delle operazioni di ricerca, inserimento, rimozione

## ■ Parametri fondamentali:

- Fattore di carico (default 0.75)
- Capacità iniziale (default 16)
- Quando il numero di elementi eccede il prodotto tra fattore di carico e la capacità della Tabella Hash, questa viene ridimensionata (ricostruita completamente da capo) in modo che la nuova capacità diventi approssimativamente il doppio della precedente
- Suggerimenti: evitare il più possibile i ridimensionamenti settando una capacità iniziale opportuna

	SEARCH		INSERT		DELETE	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array ordinati	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Liste concatenate	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Alberi BST	$O(\bar{h})$	$O(h)$	$O(\bar{h})$	$O(h)$	$O(\bar{h})$	$O(h)$
Alberi AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Tabelle Hash	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$

■ Nota:  $h = O(n)$  è l'altezza dell'albero

■  $\bar{h}$  = altezza media dell'albero